Chapter 15

Relational Query Optimization
Highlights of System R Optimizer

- **Impact:**
  - Most widely used currently; works well for < 10 joins.

- **Cost estimation:** Approximate art at best.
  - Statistics, maintained in system catalogs, used to estimate cost of operations and result sizes.
  - Considers combination of CPU and I/O costs.

- **Plan Space:** Too large, must be pruned.
  - Only the space of *left-deep plans* is considered.
    - Left-deep plans allow output of each operator to be *pipelined* into the next operator without storing it in a temporary relation.
  - Cartesian products avoided.
Overview of Query Optimization

- **Plan**: Tree of R.A. ops, with choice of alg for each op.
  - Each operator typically implemented using a `pull' interface: when an operator is `pulled' for the next output tuples, it `pulls' on its inputs and computes them.

- Two main issues:
  - For a given query, what plans are considered?
    - Algorithm to search plan space for cheapest (estimated) plan.
  - How is the cost of a plan estimated?

- **Ideally**: Want to find best plan. **Practically**: Avoid worst plans!

- We will study the System R approach.
Schema for Examples

- Sailors \((sid: \text{integer}, \ sname: \text{string}, \ rating: \text{integer}, \ age: \text{real})\)
- Reserves \((sid: \text{integer}, \ bid: \text{integer}, \ day: \text{dates}, \ rname: \text{string})\)

- Similar to old schema; \textit{rname} added for variations.
- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.
Query Blocks: Units of Optimization

- An SQL query is parsed into a collection of query blocks, and these are optimized one block at a time.
- Nested blocks are usually treated as calls to a subroutine, made once per outer tuple. (This is an oversimplification, but serves for now.)
- For each block, the plans considered are:
  - All available access methods, for each reln in FROM clause.
  - All left-deep join trees (i.e., all ways to join the relations one-at-a-time, with the inner reln in the FROM clause, considering all reln permutations and join methods.)
Relational Algebra Equivalences

- Allow us to choose different join orders and to ‘push’ selections and projections ahead of joins.
- **Selections:**  $\sigma_{c_1 \land \ldots \land c_n}(R) \equiv \sigma_{c_1}(... \sigma_{c_n}(R))$ (Cascade)
  $\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$ (Commute)
- **Projections:**  $\pi_{a_1}(R) \equiv \pi_{a_1}(... (\pi_{a_n}(R)))$ (Cascade)
- **Joins:**  $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$ (Associative)
  $(R \bowtie S) \equiv (S \bowtie R)$ (Commute)

Show that:  $R \bowtie (S \bowtie T) \equiv (T \bowtie R) \bowtie S$
More Equivalences

- A projection commutes with a selection that only uses attributes retained by the projection.
- Selection between attributes of the two arguments of a cross-product converts cross-product to a join.
- A selection on just attributes of R commutes with \( R \bowtie S \). (i.e., \( \sigma (R \bowtie S) \equiv \sigma (R) \bowtie S \))
- Similarly, if a projection follows a join \( R \bowtie S \), we can `push' it by retaining only attributes of R (and S) that are needed for the join or are kept by the projection.
Enumeration of Alternative Plans

- There are two main cases:
  - Single-relation plans
  - Multiple-relation plans

- For queries over a single relation, queries consist of a combination of selects, projects, and aggregate ops:
  - Each available access path (file scan / index) is considered, and the one with the least estimated cost is chosen.
  - The different operations are essentially carried out together (e.g., if an index is used for a selection, projection is done for each retrieved tuple, and the resulting tuples are pipelined into the aggregate computation).
Cost Estimation

- For each plan considered, must estimate cost:
  - Must estimate cost of each operation in plan tree.
    - Depends on input cardinalities.
    - We’ve already discussed how to estimate the cost of operations (sequential scan, index scan, joins, etc.)
  - Must also estimate size of result for each operation in tree!
    - Use information about the input relations.
    - For selections and joins, assume independence of predicates.
Cost Estimates for Single-Relation Plans

- Index I on primary key matches selection:
  - Cost is \( \text{Height}(I) + 1 \) for a B+ tree, about 1.2 for hash index.

- Clustered index I matching one or more selects:
  - \((\text{NPages}(I) + \text{NPages}(R)) \times \text{product of RF’s of matching selects}\).

- Non-clustered index I matching one or more selects:
  - \((\text{NPages}(I) + \text{NTuples}(R)) \times \text{product of RF’s of matching selects}\).

- Sequential scan of file:
  - \(\text{NPages}(R)\).

Note: Typically, no duplicate elimination on projections! (Exception: Done on answers if user says DISTINCT.)
Example

- If we have an index on `rating`:
  - (1/NKeys(I)) * NTuples(R) = (1/10) * 40000 tuples retrieved.
  - Clustered index: (1/NKeys(I)) * (NPages(I)+NPages(R)) = (1/10) * (50+500) pages are retrieved. (This is the cost.)
  - Unclustered index: (1/NKeys(I)) * (NPages(I)+NTuples(R)) = (1/10) * (50+40000) pages are retrieved.

- If we have an index on `sid`:
  - Would have to retrieve all tuples/pages. With a clustered index, the cost is 50+500, with unclustered index, 50+40000.

- Doing a file scan:
  - We retrieve all file pages (500).
Queries Over Multiple Relations

- Fundamental decision in System R: only left-deep join trees are considered.
  - As the number of joins increases, the number of alternative plans grows rapidly; we need to restrict the search space.
  - Left-deep trees allow us to generate all fully pipelined plans.
    - Intermediate results not written to temporary files.
    - Not all left-deep trees are fully pipelined (e.g., SM join).
Enumeration of Left-Deep Plans

- Left-deep plans differ only in the order of relations, the access method for each relation, and the join method for each join.

Enumerated using N passes (if N relations joined):

- **Pass 1:** Find best 1-relation plan for each relation.
- **Pass 2:** Find best way to join result of each 1-relation plan (as outer) to another relation. *(All 2-relation plans.)*
- **Pass N:** Find best way to join result of a (N-1)-relation plan (as outer) to the N’th relation. *(All N-relation plans.)*

For each subset of relations, retain only:

- Cheapest plan overall, plus
- Cheapest plan for each *interesting order* of the tuples.
ORDER BY, GROUP BY, aggregates etc. handled as a final step, using either an ‘interestingly ordered’ plan or an additional sorting operator.

An N-1 way plan is not combined with an additional relation unless there is a join condition between them, unless all predicates in WHERE have been used up.

- i.e., avoid Cartesian products if possible.

In spite of pruning plan space, this approach is still exponential in the # of tables.
Cost Estimation for Multirelation Plans

- Consider a query block:
  
  ```sql
  SELECT attribute list
  FROM relation list
  WHERE term1 AND ... AND termk
  ```

- Maximum # tuples in result is the product of the cardinalities of relations in the FROM clause.

- **Reduction factor (RF)** associated with each term reflects the impact of the term in reducing result size. Result cardinality = Max # tuples * product of all RF’s.

- Multirelation plans are built up by joining one new relation at a time.
  
  - Cost of join method, plus estimation of join cardinality gives us both cost estimate and result size estimate
Example

- **Pass 1:**
  - **Sailors:** B+ tree matches \( \text{rating} > 5 \), and is probably cheapest. However, if this selection is expected to retrieve a lot of tuples, and index is unclustered, file scan may be cheaper.
    - Still, B+ tree plan kept (because tuples are in \( \text{rating} \) order).
  - **Reserves:** B+ tree on \( \text{bid} = 500 \); cheapest.

- **Pass 2:**
  - We consider each plan retained from Pass 1 as the outer, and consider how to join it with the (only) other relation.
  - e.g., **Reserves as outer:** Hash index can be used to get Sailors tuples that satisfy \( \text{sid} = \) outer tuple’s \( \text{sid} \) value.
Nested Queries

- Nested block is optimized independently, with the outer tuple considered as providing a selection condition.
- Outer block is optimized with the cost of `calling’ nested block computation taken into account.
- Implicit ordering of these blocks means that some good strategies are not considered. The non-nested version of the query is typically optimized better.
Summary

- Query optimization is an important task in a relational DBMS.
- Must understand optimization in order to understand the performance impact of a given database design (relations, indexes) on a workload (set of queries).
- Two parts to optimizing a query:
  - Consider a set of alternative plans.
  - Must estimate cost of each plan that is considered.
  - Key issues: Statistics, indexes, operator implementations.
Summary (Contd.)

- **Single-relation queries:**
  - All access paths considered, cheapest is chosen.
  - **Issues:** Selections that *match* index, whether index key has all needed fields and/or provides tuples in a desired order.

- **Multiple-relation queries:**
  - All single-relation plans are first enumerated.
    - Selections/projections considered as early as possible.
  - Next, for each 1-relation plan, all ways of joining another relation (as inner) are considered.
  - Next, for each 2-relation plan that is `retained’, all ways of joining another relation (as inner) are considered, etc.
  - At each level, for each subset of relations, only best plan for each interesting order of tuples is `retained’.